Vulkan Ray Tracing – 5 New Shader Types!

Mike Bailey
mjb@cs.oregonstate.edu

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License
Analog Ray Tracing Example 😊
Digital Ray Tracing Examples

[Images of digital ray tracing examples from Blender and IronCad]
The Rasterization Shader Pipeline That You Are used to Doesn’t Apply to Vulkan Ray Tracing

- **Vertex Shader**
- **Primitive Assembly**
- **Tessellation Control Shader**
- **Tessellation Primitive Generator**
- **Tessellation Evaluation Shader**
- **Primitive Assembly**
- **Geometry Shader**
- **Primitive Assembly**
- **Rasterizer**
- **Fragment Shader**

- **Fixed Function**
- **Programmable**
The Vulkan Ray Tracing Pipeline Involves Five New Shader Types

- A **Ray Generation Shader** runs on a 2D grid of threads. It begins the entire ray-tracing operation.

- An **Intersection Shader** implements ray-primitive intersections.

- An **Any Hit Shader** is called when the Intersection Shader finds a hit. It decides if that intersection should be accepted or ignored.

- The **Closest Hit Shader** is called with the information about the hit that happened closest to the viewer. Typically, lighting is done here, or firing off new rays to handle shadows, reflections, and refractions.

- A **Miss Shader** is called when no intersections are found for a given ray. Typically, it just sets its pixel color to the background color.

Unlike the rasterization pipeline, there is no constant flow from one shader to the next. Rather, particular shaders are called to respond to particular events.

Note: none of this lives in the hardware meant for rasterization graphics. This is all built on top of the GPU compute functionality.
Example: The Ray Intersection Process for a Sphere

Sphere equation: \((x-x_c)^2 + (y-y_c)^2 + (z-z_c)^2 = R^2\)
Ray equation: \((x,y,z) = (x_0,y_0,z_0) + t(dx,dy,dz)\)

Plugging \((x,y,z)\) from the second equation into the first equation and multiplying-through and simplifying gives:

\[At^2 + Bt + C = 0\]
\[t_1, t_2 = \frac{-B \pm \sqrt{B^2 - 4AC}}{2A}\]

Solve for \(t_1, t_2\) and analyze the solution like this:

1. If both \(t_1\) and \(t_2\) are complex (i.e., have an imaginary component), then the ray missed the sphere completely.
2. If both \(t_1\) and \(t_2\) are real and identical, then the ray brushed the sphere at a tangent point.
3. If both \(t_1\) and \(t_2\) are real and different, then the ray entered and exited the sphere.

In Vulkan terms:
\[
\begin{align*}
gl\_WorldRayOrigin &= (x_0,y_0,z_0) \\
gl\_Hit &= t \\
gl\_WorldRayDirection &= (dx,dy,dz)
\end{align*}
\]
Example: The Ray Intersection Process for a Cube

Plane equation: \( Ax + By + Cz + D = 0 \)
Ray equation: \( (x,y,z) = (x_0,y_0,z_0) + t*(dx,dy,dz) \)

Plugging \((x,y,z)\) from the second equation into the first equation and multiplying through and simplifying gives:

\( Qt + R = 0 \)
Solve for \( t = -\frac{R}{Q} \)

A cube is actually the intersection of 6 half-space planes (just 4 are shown here). Each of these will produce its own \( t \) intersection value. Treat them as pairs: \((t_{x1},t_{x2}), (t_{y1},t_{y2}), (t_{z1},t_{z2})\)

The ultimate cube entry and exit values are:

\[
\begin{align*}
    t_{\text{min}} &= \max\left( \min(t_{x1}, t_{x2}), \min(t_{y1}, t_{y2}), \min(t_{z1}, t_{z2}) \right) \\
    t_{\text{max}} &= \min\left( \max(t_{x1}, t_{x2}), \max(t_{y1}, t_{y2}), \max(t_{z1}, t_{z2}) \right)
\end{align*}
\]

This algorithm works for all convex solids (e.g., cylinder, cone)
In a Raytracing, each ray typically hits a lot of Things
Acceleration Structures

- A Bottom-level Acceleration Structure (BLAS) reads the vertex data from vertex and index VkBuffers to determine bounding boxes.

- You can also supply your own bounding box information to a BLAS.

- A Top-level Acceleration Structure (TLAS) holds transformations and pointers to multiple BLASes.

- The BLAS is essentially used as a Model Coordinate bounding box, while the TLAS is used as a World Coordinate bounding box.
A “payload” is information that keeps getting passed through the processing of an individual ray. Different stages can add to it. It is finally consumed at the very end, in this case by writing `color` into the pixel being worked on.
A New Built-in GLSL Function

void trace
(
    VkAccelerationStructure topLevel, // TLAS
    uint rayFlags,
    uint cullMask,
    uint sbtRecordOffset,
    uint sbtRecordStride,
    uint missIndex,
    vec3 origin, // x₀, y₀, z₀
    float tmin, // minimum t to allow (near)
    vec3 direction, // dx, dy, dz
    float tmax, // maximum t to allow (far)
    int payload
);

In Vulkan terms (these are built-ins accessible from GLSL):
gl_WorldRayOrigin = (x₀,y₀,z₀)
gl_Hit = t
gl_WorldRayDirection = (dx,dy,dz)
Sample Intersection Shader Code

Intersect a ray with an arbitrary 3D object.
Passes data to the **Any Hit** shader.
There is a built-in ray-triangle **Intersection Shader**.

```glsl
hitAttribute vec3 attribs

void main()
{
    SpherePrimitive sph = spheres[gl_PrimitiveID];
    vec3 orig = gl_WorldRayOrigin;
    vec3 dir = normalize(gl_WorldRayDirection);
    ...
    float discr = b*b - 4.*a*c;
    if( discr < 0. )
        return;
    float tmp = (-b - sqrt(discr)) / (2.*a);
    if( gl_RayTmin < tmp && tmp < gl_RayTmax )
    {
        vec3 p = orig + tmp * dir;
        attribs = p;
        reportIntersection(tmp, 0);
        return;
    }
    tmp = (-b + sqrt(discr)) / (2.*a);
    if( gl_RayTmin < tmp && tmp < gl_RayTmax )
    {
        vec3 p = orig + tmp * dir;
        attribs = p;
        reportIntersection(tmp, 0);
        return;
    }
}
```

Intersect a ray with an arbitrary 3D object. Passes data to the **Any Hit** shader. There is a built-in ray-triangle **Intersection Shader**.
Sample Miss Shader Code

Handles a ray that doesn’t hit *any* objects

```glsl
Layout( location=0 ) rayPayload
{
    vec4 color;
}
myPayload;

void main( )
{
    myPayload.color = vec4( 0., 0., 0., 1. );
}
```

![Diagram showing ray generation, traversal, and hit shader processes](Diagram.png)
Sample Any Hit Shader Code

Handle a ray that hits *anything*. Store information on each hit. Can reject a hit.

```cpp
layout( binding = 4, set = 0) buffer outputProperties
{
    float outputValues[ ];
} outputData;

layout(location = 0) rayPayloadIn uint outputId;
layout(location = 1) rayPayloadIn uint hitCounter;
hitAttribute vec3 attribs;

void main()
{
    outputData.outputValues[ outputId + hitCounter ] = gl_PrimitiveID;
    hitCounter = hitCounter + 1;
}
```
Sample Closest Hit Shader

Handle the intersection closest to the viewer.
Collects data from calls to the Any Hit shader.
This shader can spawn more rays to handle shadows, reflections, and refractions.

uniform sampler2D uTexUnit;

rayPayload myPayload
{
    vec4 color;
};

void
main( )
{
    vec3 stp = gl_WorldRayOrigin + gl_Hit * gl_WorldRayDirection;
    color = texture( uTexUnit, stp );  // material properties lookup
}

In Vulkan terms:
gl_WorldRayOrigin = (x₀,y₀,z₀)
gl_Hit = t
gl_WorldRayDirection = (dx,dy,dz)
Other New Built-in Functions

void \texttt{terminateRay}();

\textbf{Loosely equivalent to “discard”}

void \texttt{ignoreIntersection}();

void \texttt{reportIntersection} ( float hit, uint hitKind );
The Trigger comes from the Command Buffer: `vlCmdBindPipeline( )` and `vkCmdTraceRays( )`

```c
vkCmdBindPipeline( CommandBuffer, VK_PIPELINE_BIND_POINT_RAY_TRACING, RaytracePipeline );

vkCmdTraceRays( CommandBuffer, raygenShaderBindingTableBuffer, raygenShaderBindingOffset, missShaderBindingTableBuffer, missShaderBindingOffset, missShaderBindingStride, hitShaderBindingTableBuffer, hitShaderBindingOffset, hitShaderBindingStride, callableShaderBindingTableBuffer, callableShaderBindingOffset, callableShaderBindingStride, width, height, depth );
```
Check This Out!

https://www.youtube.com/watch?v=QL7sXc2iNJ8